Synthesizing Network Requirements Using Parallel Scientific Applications*

Anand Sivasubramaniam
Aman Singla
Umakishore Ramachandran
H. Venkateswaran

Technical Report GIT-CC-94/31
July 1994

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
Phone: (404) 894-5136
Fax: (404) 894-9442
e-mail: rama@cc.gatech.edu

Abstract

We synthesize the link bandwidth requirement for a binary hypercube topology using a set of five scientific applications. We use an execution-driven simulator called SPASM to collect data points for system sizes that are feasible to be simulated. These data points are then used in a regression analysis for projecting the link bandwidth requirements for larger systems. These requirements are projected as a function of the following system parameters: number of processors, CPU clock speed, and problem size. These results are also used to project the link bandwidths for other network topologies. A significant contribution of our study is in quantifying the link bandwidth that has to be made available to tolerate a given amount of network overhead in an application. Our results show that typical link bandwidths (200-300 MBytes/sec) found in current commercial parallel architectures (such as Intel Paragon and Cray T3D) would have fairly low network overhead for the scientific applications considered in this study. For two of the applications, this overhead is negligible. For the other applications, this overhead is about 30% of the execution time provided the problem sizes are increased commensurate with the processor clock speed.

Key Words: Interconnection Networks, Latency, Contention, Execution-driven Simulation, Application-driven Study.

*This work has been funded in part by NSF grants MIPS-9058430 and MIPS-9200005, and an equipment grant from DEC.

1 Introduction

Parallel machines promise to solve computationally intensive problems that may not be feasibly computed due to resource limitations on sequential machines. Despite this promise, the delivered performance of these machines often falls short of the projected peak performance. The disparity between the expected and observed performance may be due to application deficiencies such as serial fraction and work-imbalance, software slow-down due to scheduling and runtime overheads, and hardware slow-down stemming from the synchronization and communication requirements in the application. For building a general-purpose parallel machine, it is essential to identify and quantify the architectural requirements necessary to assure good performance over a wide range of applications. Such a synthesis of requirements from an application view-point can help us make cost vs. performance trade-offs in important architectural design decisions. The network is an important artifact in a parallel machine limiting the performance of many applications, and is the focus of this study. Using five scientific applications, we quantify bandwidth requirements needed to limit the overheads arising from the network to an acceptable level for these applications.

Latency and *contention* are two defining characteristics of a network from the application viewpoint. Latency is the sum of the time spent in transmitting a message over the network links and the switching time assuming that the message did not have to contend for any network links. Contention is the time spent by a message in the network waiting for links to become free. Both latency and contention depend on a number of factors including the connectivity, the bandwidth of the links in the network, the switching delays, and the length of the message. Of the architectural factors, link bandwidth and connectivity are the most crucial network parameters impacting latency and contention. Hence, in order to quantify requirements that limit network overheads (latency and contention) to an acceptable level, it is necessary to study the impact of link bandwidth and network connectivity on these overheads.

Dally [10] and Agarwal [2] present analytical models to study the impact of network connectivity and link bandwidth for k-ary n-cube networks. The results suggest that low-dimensional networks are preferred (based on physical and technological constraints) when the network contention is ignored or when the workload (the application) exhibits sufficient network locality; and that higher dimensional networks may be needed otherwise. Adve and Vernon [1] show using analytical models that network locality has an important role to play in the performance of the mesh. Since network requirements are sensitive to the workload, it is necessary to study them in the context of real applications.

The RISC ideology clearly illustrates the importance of using real applications in synthesizing architectural requirements. Several researchers have used this approach for parallel architectural studies [22, 9, 15]. Cypher et al. [9] use a range of scientific applications in quantifying the processing, memory, communication and I/O requirements. They present the communication requirements in terms of the number of messages exchanged between processors and the volume (size) of these messages. As identified in [24], communication in parallel applications may be categorized by the following attributes: communication volume, the communication pattern, the communication frequency and the ability to overlap communication with computation. A static analysis of the communication as conducted in [9] fails to capture the last two attributes, making it very difficult to quantify the contention in the system.

The importance of simulation in capturing the dynamics of parallel system[1] behavior has been clearly illustrated in [14, 24, 26]. In particular, using an execution-driven simulation, one can faithfully capture all the attributes of communication that are important to network requirements synthesis. For example, in [14] the authors use an execution-driven simulator to study k-ary n-cube networks in the context of applications drawn from image understanding, and show the impact of application characteristics on the choice of the network topology. We take a similar approach to deriving the network requirements in this study.

Using an execution-driven simulation platform called SPASM [27, 26], we simulate the execution of five scientific applications on an architectural platform with a binary hypercube network topology. We vary the link bandwidth on the hypercube and quantify its impact on application performance. From these results, we arrive at link bandwidths that are needed to limit network overheads to an acceptable level. We also study the impact of the number of processors, the CPU clock speed and the application problem size on bandwidth requirements. Using regression analysis and analytical techniques, we extrapolate requirements for larger systems of 1024 processors and other network topologies. The results suggest that existing link bandwidth of 200-300 MBytes/sec available on machines like Intel Paragon [16] and Cray T3D [19] can easily sustain the requirements of two applications (EP and FFT) even on high-speed processors of the future. For the other three, one may be able to maintain network overheads at an acceptable level if the problem size is increased commensurate with the processing speed. Section 2 gives an overview of our methodology and details of the simulation platform; section 3 briefly describes the hardware platform and applications used in this study; section 4 presents results from our experiments; section 5 summarizes the implication of these results; and section 6 presents concluding remarks.

2 Methodology

As observed in [24], communication in an application may be characterized by four attributes. Volume refers to the number and size of messages. The communication pattern in the application determines the source-destination pairs for the messages, and reflects on the application's ability to exploit network locality. Frequency pertains to the temporal aspects of communication, i.e., the interval between successive network accesses by each processor as well as the interleaving in time of accesses from different processors. This temporal aspect of communication would play an important role in determining network contention. Tolerance is the ability of an application to hide network overheads by overlapping computation with communication. Modeling all these attributes of communication in a parallel application is extremely difficult by simple static analysis. Further, the dynamic access patterns exhibited by many applications makes modeling more complex. Several researchers [24, 26, 14] have observed that simulation and profiling tools are useful for capturing the communication behavior of applications.

In this study, we use an execution-driven simulator called SPASM (Simulator for Parallel Architectural Scalability Measurements) that enables us to accurately model the behavior of applications on a number of simulated hardware platforms. SPASM has been written using CSIM [18], a process oriented sequential simulation package, and currently runs on SPARCstations. The input to the simulator are parallel applications written in C. These programs are preprocessed

[1] The term, parallel system, is used to denote an application-architecture combination.

(to label shared memory accesses), the compiled assembly code is augmented with cycle counting instructions, and the assembled binary is linked with the simulator code. As with other recent simulators [6, 11, 7, 21], bulk of the instructions is executed at the speed of the native processor (the SPARC in this case) and only instructions (such as LOADs and STOREs on a shared memory platform or SENDs and RECEIVEs on a message-passing platform) that may potentially involve a network access are simulated. The reader is referred to [27, 26] for a detailed description of SPASM where we illustrated its use in studying the scalability of a number of parallel applications on different shared memory [26] and message-passing [27] platforms. The input parameters that may be specified to SPASM are the number of processors, the CPU clock speed, the network topology, the link bandwidth and switching delays. We can thus vary a range of system parameters and study their impact on application performance.

SPASM provides a wide range of statistical information about the execution of the program. The algorithmic overhead (arising from factors such as the serial part and work-imbalance in the algorithm) and the network overheads (latency and contention) are the important overheads that are of relevance to this study. The profiling capabilities of SPASM (outlined in [26]) provide a novel isolation and quantification of these overheads that contribute to the performance of the parallel system. It gives the *total time* (simulated time) which is the maximum of the running times of the individual parallel processors. This is the time that would be taken by an execution of the parallel program on the target parallel machine. SPASM also gives the *ideal time*, which is the time taken by the parallel program to execute on an ideal machine such as the PRAM [30]. This metric includes the algorithmic overheads but does not include any overheads arising from architectural limitations. Of the network overheads, the time that a message would have taken in a contention free environment is charged to the latency overhead, while the rest of the time is accounted for in the contention overhead.

To synthesize the communication requirements of parallel applications, the separation of the overheads provided by SPASM is crucial. For instance, an application may have an algorithmic deficiency due to either a large serial part or due to work-imbalance, in which case 100% efficiency[2] is impossible regardless of other architectural parameters. The separation of overheads, provided by SPASM, enables us to quantify the bandwidth requirements as a function of acceptable network overheads (latency and contention).

3 Experimental Setup

We have chosen a CC-NUMA (Cache Coherent Non-Uniform Memory Access) shared memory multiprocessor as the architectural platform for this study. Since uniprocessor architecture is getting standardized with the advent of RISC technology, we fix most of the processor characteristics by using a SPARC chip as the baseline for each processor in a parallel system. But to study the impact of processor speed on network requirements we allow the clock speed of a processor to be varied. Each node in the system has a piece of the globally shared memory and a 2-way set-associative private cache (64KBytes with 32 byte blocks). The cache is maintained sequentially consistent using an invalidation-based fully-mapped directory-based cache coherence scheme. Rothberg et al. [22] show that a cache of moderate size (64KBytes) suffices to capture the working set in many applications, and Wood et al. [29] show that the network traffic generated is

[2] *Efficiency* is defined as $speedup(p)/p$ where p is the number of processors. *Speedup(p)* is the ratio of the time taken to execute the parallel application on 1 processor to the time taken to execute the same on p processors.

not significantly different across cache coherence protocols over a wide range of applications. Thus in our approach to synthesizing network requirements, we fix the cache parameters and vary only the clock speed of the processor and the network parameters. The synchronization primitive supported in hardware is a *test-and-set* operation and applications use a *test-test-and-set* to implement higher level synchronization.

The study is conducted for a binary hypercube interconnect. The hypercube is assumed to have serial (1-bit wide) unidirectional links and uses the e-cube routing algorithm [28]. Messages are circuit-switched using a wormhole routing strategy and the switching delay is assumed to be zero. Ideally, we would like to simulate other networks as well in order to study the change in link bandwidth requirements with network connectivity. Since these simulations take an inordinate amount of time, we have restricted ourselves to simulating the hypercube network in this study. We use the results from the hypercube study in hypothesizing the requirements for other networks using analytical techniques coupled with application knowledge.

We have chosen five parallel scientific applications in this study. Three of the applications (EP, IS and CG) are from the NAS parallel benchmark suite [5]; CHOLESKY is from the SPLASH benchmark suite [25]; and FFT is the well-known Fast Fourier Transform algorithm. EP and FFT are well-structured applications with regular communication patterns determinable at compile-time, with the difference that EP has a higher computation to communication ratio. IS also has a regular communication pattern, but in addition it uses locks for mutual exclusion during the execution. CG and CHOLESKY are different from the other applications in that their communication patterns are not regular (both use sparse matrices) and cannot be determined at compile time. While a certain number of rows of the matrix in CG is assigned to a processor at compile time (static scheduling), CHOLESKY uses a dynamically maintained queue of runnable tasks. The Appendix gives further details on the applications.

4 Performance Results

In this section, we present results from our simulation experiments. Using these results, we quantify the link bandwidth necessary to support the efficient performance of these applications and project the bandwidth requirements for building large-scale parallel systems with a binary hypercube topology.

The experiments have been conducted over a range of processors (p=4, 8, 16, 32, 64), CPU clock speeds (s=33, 100, 300 MHz) and link bandwidths (b=1, 20, 100, 200, 600 and 1000 MBytes/sec). The problem size n of the applications has been varied as 16K, 32K, 64K, 128K and 256K for EP, IS and FFT, 1400 $\times$ 1400 and 5600 $\times$ 5600 for CG, and 1806 $\times$ 1806 for CHOLESKY. In studying the effect of each parameter (processors, clock speed, problem size), we keep the other two constant.

4.1 Impact of System Parameters on Bandwidth Requirements

As the link bandwidth is increased, the efficiency of the system is also expected to increase as shown in Figure 1. But we soon reach a point of diminishing returns beyond which increasing the bandwidth does not have a significant impact on application performance (the curves flatten) since the network overheads (both latency and contention) are sufficiently

low at this point. In all our results, we observe such a distinct knee. One would expect the efficiency beyond this knee to be close to 100%. But owing to algorithmic overheads such as serial part or work-imbalance, the knee may occur at a much lower efficiency (e_0 in Figure 1). These algorithmic overheads may also cause the curves for each configuration of system parameters to flatten out at entirely different levels in the efficiency spectrum. The bandwidth corresponding to the knee (b_0) still represents an ideal point at which we would like to operate since the network overheads beyond this knee are minimal. We track the horizontal movement of this knee to study the impact of system parameters (processors, CPU clock speed, problem size) on link bandwidth requirements.

Figures 3, 4, 5, 6 and 7 show the impact of varying link bandwidth on the efficiency of EP, IS, FFT, CG and CHOLESKY respectively, across different number of processors (p). The knees for EP and FFT, which display a high computation to communication ratio, occur at low bandwidths and are hardly noticeable in these figures. The algorithmic overheads in these applications is negligible yielding efficiencies that are close to 100%. For the other applications, the knee occurs at a higher bandwidth. Further, the curves tend to flatten at different efficiencies suggesting the presence of algorithmic overheads. For all the applications, the knee shifts to the right as the number of processors is increased indicating the need for higher bandwidth. As the number of processors is increased, the network accesses incurred by a processor in the system may increase or decrease depending on the application, but each such access would incur a larger overhead from contending for network resources (due to the larger number of messages in the network as a whole). Further, the computation performed by a processor is expected to decrease, lowering the computation to communication ratio, thus making the network requirements more stringent.

Figures 8, 9, 10, 11 and 12 show the impact of link bandwidth on the efficiency of EP, IS, FFT, CG and CHOLESKY respectively, across different CPU clock speeds (s). As the CPU clock speed is increased, the computation to communication ratio decreases. In order to sustain the same efficiency, communication has to be sped up to keep pace with the CPU speed thus shifting the knee to the right uniformly across all applications.

Figures 13, 14, 15, and 16 show the impact of link bandwidth on the efficiency of EP, IS, FFT, and CG respectively, across different problem sizes. An increase in problem size is likely to increase the amount of computation performed by a processor. At the same time, a larger problem may also increase the network accesses incurred by a processor. In EP, FFT and CG, the former effect is more dominant thereby increasing the computation to communication ratio, making the knee move to the left as the problem size is increased. The two counteracting effects nearly compensate each other in IS showing negligible shift in the knee.

4.2 Quantifying Link Bandwidth Requirements

We analyze bandwidth requirements using the above simulation results in projecting requirements for large-scale parallel systems. We track the change in the knee with system parameters by quantifying the link bandwidth needed to limit the network overheads to a certain fraction of the overall execution time. This fraction would determine the closeness of the operating point to the knee. Ideally, one would like to operate as close to the knee as possible. But owing to either cost or technological constraints, one may be forced to operate at a lower bandwidth and it would be interesting to investigate if

one may still obtain reasonable efficiencies under these constraints. With the ability to tolerate a larger network overhead, the bandwidth requirements are expected to decrease as shown by the curve labelled "Actual" in Figure 2. To calculate the bandwidth requirement needed to limit the network overhead (both the latency and contention component) to a certain value, we simulate the applications over a range of link bandwidths. We perform a linear interpolation between these data points as shown by the curve labelled "Simulated" in figure 2. Using these points, we calculate the bandwidth (b_s) required to limit the network overhead to $x\%$ of the total execution time. This bandwidth would represent a good upper bound on the corresponding "Actual" bandwidth (b_a) required. In the following discussions, we present requirements for x = 10%, 30% and 50%. These requirements are expected to change with the number of processors, the CPU clock speed and the application problem size. The rate of change in the knee is used to study the impact of these system parameters. In cases where the analysis is simple, we use our knowledge of the application and architectural characteristics in extrapolating the performance for larger systems. In cases where such a static analysis is not possible (due to the dynamic nature of the execution), we perform a non-linear regression analysis of the simulation results using a multivariate secant method with a 95% confidence interval in the SAS [23] statistics package.

Using this methodology, we now discuss for each application its intrinsic characteristics that impact the communication and computation requirements; present the link bandwidth requirements as a function of increasing number of processors, CPU clock speed, and problem size; and project the requirements for a 1024-node system with a problem size appropriate for such a system.

EP

EP has a high computation to communication ratio with the communication being restricted to a few logarithmic global sum operations. The bulk of the time is spent in local computation and as a result, even a bandwidth of 1 MByte/sec is adequate to limit network overheads to less than 10% of the execution time (see Table 1). As the number of processors is increased, the communication incurred by a processor in the global sum operation grows logarithmically and a bandwidth of 10 MBytes/sec can probably sustain even a system with 1024 processors. As the clock speed is increased, the time spent by a processor in the local computation is expected to decrease linearly and the bandwidth requirement for the global sum operation needs to increase at the same rate in order to maintain the same efficiency. Table 2 reflects this behavior. As the problem size (n) is increased for this application, the local computation incurred by a processor is expected to grow as $O(n)$ while the communication (both the number of global sum operations as well as the number of messages for a single operation) remains the same. As a result, the bandwidth requirements are expected to decrease linearly with problem size. Given that real world problem sizes for this application are of the order of $n = 2^{28}$ [5], a very low link bandwidth (less than 1 MByte/sec) would suffice to yield an efficiency close to 100%.

IS

IS is more communication intensive than EP and its bandwidth requirements are expected to be considerably higher. The appendix gives details on the characteristics of this application, and there are two dominant phases in the execution that account for the bulk of the communication. In the first, a processor accesses a part of the local buckets of every other

processor in making updates to the global buckets allotted to it. In the second phase, a processor locks each of the global buckets (that is partitioned by consecutive chunks across processors) in ranking the list of elements allotted to it. With an increase in the number of processors, a processor needs to access data from more processors in the former phase. In the latter, the amount of global buckets that is allocated to a processor decreases linearly with increase in processors due to the partitioning scheme. Hence, in both these phases, the communication is expected to grow as $O(p)$ with increase in processors. Further, the computation performed by a processor decreases with an increase in processors, but the rate is less than linear owing to algorithmic deficiencies in the problem [26]. These factors combine to yield a considerable bandwidth requirement for larger systems (see Table 3), if we are willing to tolerate less than 10% network overheads. As the CPU clock speed is increased, the computation to communication ratio decreases, making the requirements more stringent as shown in Table 4. As the problem size (n) is increased, the communication in the above mentioned phases increases linearly. The local computation also increases, but the former effect is more prominent as is shown in Table 5, where the bandwidth requirements grow moderately with problem size.

Using these results, the bandwidth requirements for IS are projected in Table 6 for a 1024 node system and a problem size of 2^{23} that is representative of a real world problem [5]. This table shows that bandwidth requirements of IS are considerably high. We may at best be able to operate currently at around 50% network overhead range with 33 MHz processors given that link bandwidth of state-of-the-art networks is around 200-300 MBytes/sec. With faster processors like the DEC Alpha, the network becomes an even bigger bottleneck for this application.

In projecting the above bandwidth requirements for this application with 1024 processors, both the number of buckets as well as the number of list elements have been increased for the larger problem. But bucket sort is frequently used in cases where the number of buckets is relatively independent of the number of elements in the list to be sorted. A scaling strategy where the size of the list is increased and the number of buckets is maintained constant would cause no change in communication in the above mentioned phases of IS, while the computation is expected to grow as $O(n)$. Hence, if we employ such a scaling strategy and increase the problem size linearly with the CPU clock speed, we may be able to limit the network overheads to within 30-50% for this application with existing technology.

FFT

The implementation of FFT has been optimized so that all the communication takes place in one phase where every processor communicates with every other processor, and the communication in this phase is skewed in order to reduce contention. The computation performed by a processor in FFT grows as $O((n \log n)/p)$ while the communication grows as $O(n(p-1)/p^2)$. Thus, these components decrease at comparable rates with an increase in the number of processors. As the number of processors is increased, the contention encountered by each message in the network is expected to grow. However, due to the implementation strategy the bandwidth requirements of the network grow slowly with the number of processors as is shown in Table 6. These requirements can be satisfied even for faster processors (see Table 8). As we mentioned earlier, the computation to communication ratio is proportional to $O(\log n)$, and the network requirements are expected to become even less stringent as the problem size is increased. Table 9 confirms this observation. Hence,

in projecting the requirements for a 1024-node system, link bandwidths of around 100-150 MBytes/sec would suffice to limit the network overheads to less than 10% of the execution time (see Table 10). The results shown in the above tables agree with theoretical results presented in [12] where the authors show that FFT is scalable on the cube topology and the achievable efficiency is only limited by the ratio of the CPU clock speed and the link bandwidth.

CG

The main communication in CG occurs in the multiplication of a sparse matrix with a dense vector (see the appendix). Each processor performs this operation for a contiguous set of rows allocated to it. The elements of the vector that are needed by a processor to perform this operation depend on the distribution of non-zero elements in the matrix and may involve external accesses. Once an element is fetched, a processor may reuse it for a non-zero element in another row at the same column position. As the number of processors is increased, the number of rows allocated to a processor decreases thus decreasing the computation that it performs. Increasing the number of processors has a dual impact on communication. Since the number of rows that need to be computed decreases, the probability of external accesses decreases. There is also a decreased probability of reusing a fetched data item for computing another row. These complicated interactions are to a large extent dependent on the input data and are difficult to analyze statically. We use the data sets supplied with the NAS benchmarks [5]. The results from our simulation are given in Table 11. We observe that the effect of lower local computation, and lesser data reuse has a more significant impact in increasing the communication requirements for larger systems. Increasing the clock speed has an almost linear impact on increasing bandwidth requirements as given in Table 12. As the problem size is increased, the local computation increases, and the probability of data re-use also increases. The rate at which these factors impact the requirements depends on the sparsity factor of the matrix. Table 13 shows the requirements for two different problem sizes. For the 1400×1400 problem, the sparsity factor is 0.04, while the sparsity factor for the 5600×5600 problem is 0.02. The corresponding factor for the 14000×14000 problem suggested in [5] is 0.1 and we scale down the bandwidth requirements accordingly in Table 14 for a 1024 node system. The results suggest that we may be able to limit the overheads to within 50% of the execution time with existing technology. As the processors get faster than 100 MHz, it would need a considerable amount of bandwidth to limit overheads to within 30%. But with faster processors, and larger system configurations, one may expect to solve larger problems as well. If we increase the problem size (number of rows of the matrix) linearly with the clock speed of the processor, one may expect the bandwidth requirements to remain constant, and we may be able to limit network overheads to within 30% of execution time even with existing technology.

CHOLESKY

This application performs a Cholesky factorization of a sparse positive definite matrix (see the appendix). Each processor while working on a column will need to access the non-zero elements in the same row position of other columns. Once a non-local element is fetched, the processor can reuse it for the next column that it has to process. The communication pattern in processing a column is similar to that of CG. The difference is that the allocation of columns to processors in CHOLESKY is done dynamically. As with CG, an increase in the number of processors is expected to decrease

the computation to communication ratio as well as the probability of data reuse. Further, the network overheads for implementing dynamic scheduling are also expected to increase for larger systems. Table 15 reflects this trend, showing that bandwidth requirements for CHOLESKY grow modestly with increase in processors. Still, the requirements may be easily satisfied with existing technology for 1024 processors. Even with a 300 MHz clock, one may be able to limit network overheads to around 30% as shown in Table 16. Owing to resource constraints, we have not been able to simulate other problem sizes for CHOLESKY in this study. But an increase in problem size is expected to increase the the computation to communication ratio and has been experimentally verified on the KSR-1, which suggests that bandwidth requirements are expected to decrease with problem size. Hence, as the processors get faster, one may still be able to maintain network overheads at an acceptable level with existing technology if the problem size is increased correspondingly.

5 Discussion

In the previous section, we quantified the link bandwidth requirements of five applications for the binary hypercube topology as a function of the number of processors, CPU clock speed and problem size. Based on these results we also projected the requirements of large systems built with 1024 processors and CPU clock speeds upto 300 MHz. We observed that EP has negligible bandwidth requirements and FFT has moderate requirements that can be easily sustained. The network overheads for CG and CHOLESKY may be maintained at an acceptable level for current day processors, and as the processor speed increases, one may still be able to tolerate these overheads provided the problem size is increased commensurately. The network overheads of IS are tolerable for slow processors, but the requirements become unmanageable as the clock speed increases. As we observed, the deficiency in this problem is in the way the problem is scaled (the number of buckets is scaled linearly with the size of the input list to be sorted). On the other hand, if the number of buckets is maintained constant, it may be possible to sustain bandwidth requirements by increasing the problem size linearly with the processing speed.

In [20], the authors show that the applications EP, IS, and CG scale well on a 32-node KSR-1. In our study, we use the same implementations of these applications to synthesize the network requirements. Although our results suggest that these applications may incur overheads affecting their scalability, this does not contradict the results presented in [20] since the implications of our study are for larger systems built with much faster processors.

All of the above link bandwidth results have been presented for the binary hypercube network topology. The cube represents a highly scalable network where the bisection bandwidth grows linearly with the number of processors. Even though cubes of 1024 nodes have been built [13], cost and technology factors often play an important role in its physical realization. Agarwal [2] and Dally [10] show that wire delays (due to increased wire lengths associated with planar layouts) of higher dimensional networks make low dimensional networks more viable. The 2-dimensional [17] and 3-dimensional [19, 3] toroids are common topologies used in current day networks, and it would be interesting to project link bandwidth requirements for these topologies.

A metric that is often used to compare different networks is the cross-section bandwidth available per processor. On a k-ary n-cube, the cross-section bandwidth available per processor is inversely proportional to the radix k of the

network. One may use a simple rule of thumb of maintaining per processor cross-section bandwidth constant. For example, considering a 1024-node system, the link bandwidth requirement for a 32-ary 2-cube would be 16 times the 2-ary 10-cube bandwidth; similarly the requirement for a 3-D network would be around 5 times the 10-D network. Such a projection assumes that the communication in an application is devoid of any network locality and that each message crosses the bisection. But we know that applications normally tend to exploit network locality and the projection can thus become very pessimistic [1]. With a little knowledge about the communication behavior of applications, one may be able to reduce the degree of pessimism. In both FFT and IS, every processor communicates with every other processor, and thus only 50% of the messages cross the bisection. Similarly, instrumentation in our simulation showed that around 50% of the messages in CG and CHOLESKY traverse the bisection. To reduce the degree of pessimism in these projections, one may thus introduce a correction factor of 0.5 that can be multiplied with the above-mentioned factors of 16 and 5 in projecting the bandwidths for 2-D and 3-D networks respectively. EP would still need negligible bandwidth and we can still limit network overheads of FFT to around 30% on these networks with existing technology. But the problem sizes for IS, CG and CHOLESKY would have to grow by a factor of 8 compared to their cube counterparts if we are to sustain the efficiency attainable on a 2-D network with current technology. Despite the correction factor, these projections are still expected to be pessimistic since the method ignores the temporal aspect of communication. The projection assumes that every message in the system traverses the bisection at the same time. If the message pattern is temporally skewed, then a lower link bandwidth may suffice for a given network overhead. It is almost impossible to determine these skews statically, especially for applications like CG and CHOLESKY where the communication pattern is dynamic. It would be interesting to conduct a detailed simulation for these network topologies to confirm these projections.

6 Concluding Remarks

In this study, we undertook the task of synthesizing the network requirements of five parallel scientific applications. Such a study can help in making cost-performance trade-offs in designing and implementing networks for large scale multiprocessors. We used an execution-driven simulator called SPASM for simulating the applications on an architectural platform with a binary hypercube topology. Simulation can faithfully capture all the attributes of communication which are relatively difficult to be modeled by a simple static analysis of the applications. The link bandwidth of the simulated platform was varied and its impact on application performance was quantified. From these results, the link bandwidth requirements for limiting the network overheads to an acceptable level were identified. We also studied the impact of system parameters (number of processors, processing speed, problem size) on link bandwidth requirements. Using regression analysis and analytical techniques, we projected requirements for large scale parallel systems with 1024 processors and other network topologies. The results show that existing link bandwidth of 200-300 MBytes/sec available on machines like Intel Paragon [16] and Cray T3D [19] can sustain high speed applications with fairly low network overhead. For applications like EP and FFT, this overhead is negligible. For the other applications, this overhead is about 30% of the execution time provided the problem sizes are increased commensurate with the processor clock speed.

References

[1] V. S. Adve and M. K. Vernon. Performance analysis of mesh interconnection networks with deterministic routing. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):225–246, March 1994.

[2] A. Agarwal. Limits on Interconnection Network Performance. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):398–412, October 1991.

[3] R. Alverson et al. The Tera Computer System. In *Proceedings of the ACM 1990 International Conference on Supercomputing*, pages 1–6, Amsterdam, Netherlands, 1990.

[4] T. E. Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[5] D. Bailey et al. The NAS Parallel Benchmarks. *International Journal of Supercomputer Applications*, 5(3):63–73, 1991.

[6] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. PROTEUS : A high-performance parallel-architecture simulator. Technical Report MIT-LCS-TR-516, Massachusetts Institute of Technology, Cambridge, MA 02139, September 1991.

[7] R. G. Covington, S. Madala, V. Mehta, J. R. Jump, and J. B. Sinclair. The Rice parallel processing testbed. In *Proceedings of the ACM SIGMETRICS 1988 Conference on Measurement and Modeling of Computer Systems*, pages 4–11, Santa Fe, NM, May 1988.

[8] D. Culler et al. LogP : Towards a realistic model of parallel computation. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, May 1993.

[9] R. Cypher, A. Ho, S. Konstantinidou, and P. Messina. Architectural requirements of parallel scientific applications with explicit communication. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 2–13, May 1993.

[10] W. J. Dally. Performance analysis of k-ary n-cube interconnection networks. *IEEE Transactions on Computer Systems*, 39(6):775–785, June 1990.

[11] H. Davis, S. R. Goldschmidt, and J. L. Hennessy. Multiprocessor Simulation and Tracing Using Tango. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages II 99–107, 1991.

[12] A. Gupta and V. Kumar. The scalability of FFT on parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(8):922–932, August 1993.

[13] J. L. Gustafson, G. R. Montry, and R. E. Benner. Development of Parallel Methods for a 1024-node Hypercube. *SIAM Journal on Scientific and Statistical Computing*, 9(4):609–638, 1988.

[14] W. B. Ligon III and U. Ramachandran. Simulating interconnection networks in RAW. In *Proceedings of the Seventh International Parallel Processing Symposium*, April 1993.

[15] W. B. Ligon III and U. Ramachandran. Evaluating multigauge architectures for computer vision. *Journal of Parallel and Distributed Computing*, 21:323–333, June 1994.

[16] Intel Corporation, Oregon. *Paragon User's Guide*, 1993.

[17] D. Lenoski, J. Laudon, K. Gharachorloo, W-D Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford DASH multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[18] Microelectronics and Computer Technology Corporation, Austin, TX 78759. *CSIM User's Guide*, 1990.

[19] W. Oed. The Cray Research Massively Parallel Processor System Cray T3D, 1993.

[20] U. Ramachandran, G. Shah, S. Ravikumar, and J. Muthukumarasamy. Scalability study of the KSR-1. In *Proceedings of the 1993 International Conference on Parallel Processing*, pages I–237–240, August 1993.

[21] S. K. Reinhardt et al. The Wisconsin Wind Tunnel : Virtual prototyping of parallel computers. In *Proceedings of the ACM SIGMETRICS 1993 Conference on Measurement and Modeling of Computer Systems*, pages 48–60, Santa Clara, CA, May 1993.

[22] E. Rothberg, J. P. Singh, and A. Gupta. Working sets, cache sizes and node granularity issues for large-scale multiprocessors. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 14–25, May 1993.

[23] SAS Institute Inc., Cary, NC 27512. *SAS/STAT User's Guide*, 1988.

[24] J. P. Singh, E. Rothberg, and A. Gupta. Modeling communication in parallel algorithms: A fruitful interaction between theory and systems? In *Proceedings of the Sixth Annual ACM Symposium on Parallel Algorithms and Architectures*, 1994.

[25] J. P. Singh, W-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical Report CSL-TR-91-469, Computer Systems Laboratory, Stanford University, 1991.

[26] A. Sivasubramaniam, A. Singla, U. Ramachandran, and H. Venkateswaran. An Approach to Scalability Study of Shared Memory Parallel Systems. In *Proceedings of the ACM SIGMETRICS 1994 Conference on Measurement and Modeling of Computer Systems*, pages 171–180, May 1994.

[27] A. Sivasubramaniam, A. Singla, U. Ramachandran, and H. Venkateswaran. A Simulation-based Scalability Study of Parallel Systems. *Journal of Parallel and Distributed Computing*, 1994. To appear.

[28] H. Sullivan and T. R. Bashkow. A large scale, homogenous, fully-distributed parallel machine. In *Proceedings of the 4th Annual Symposium on Computer Architecture*, pages 105–117, March 1977.

[29] D. A. Wood et al. Mechanisms for cooperative shared memory. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 156–167, May 1993.

[30] J. C. Wyllie. *The Complexity of Parallel Computations*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, NY, 1979.

Impact of Number of Processors

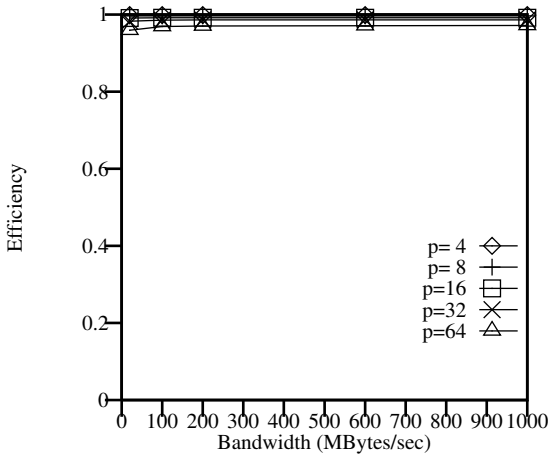

Figure 3: EP

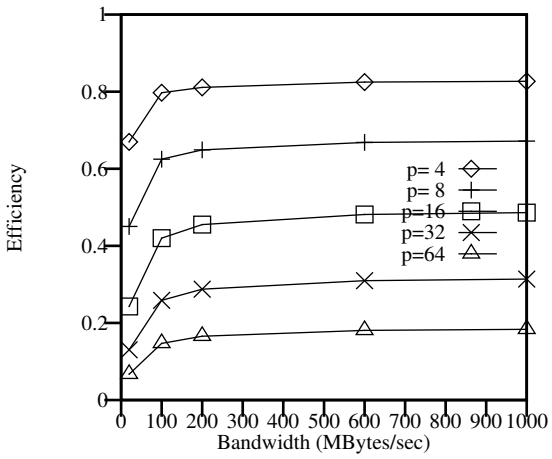

Figure 4: IS

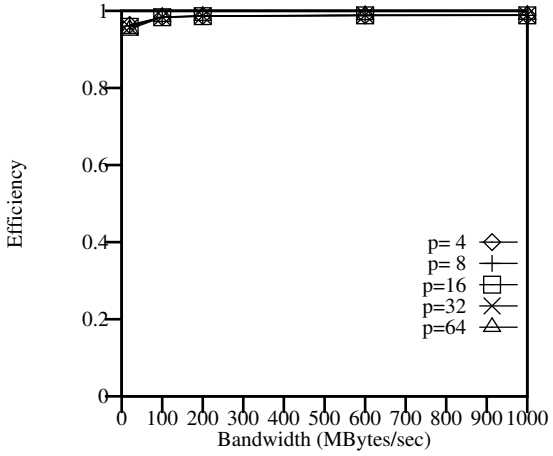

Figure 5: FFT

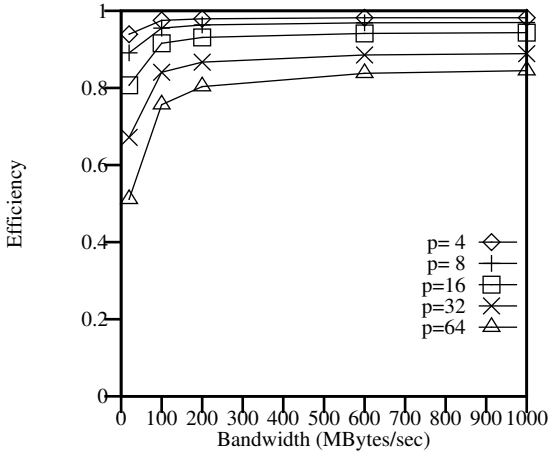

Figure 6: CG

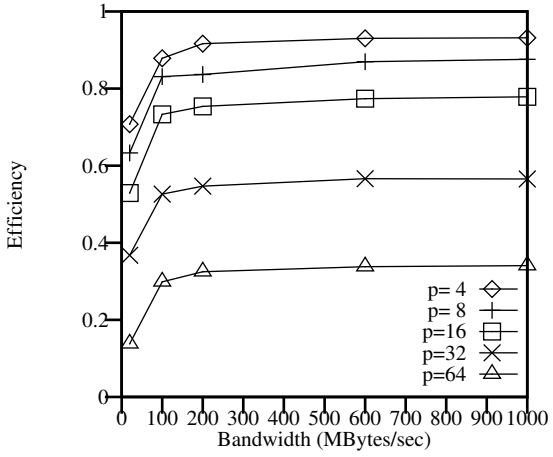

Figure 7: CHOLESKY

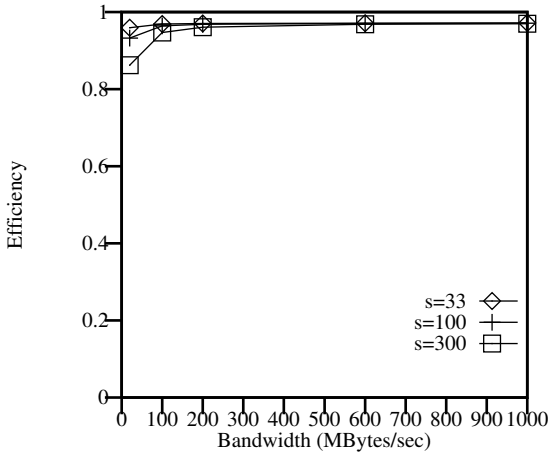

Figure 8: EP

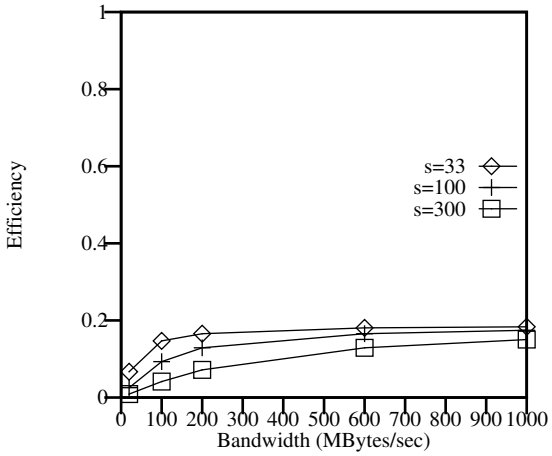

Figure 9: IS

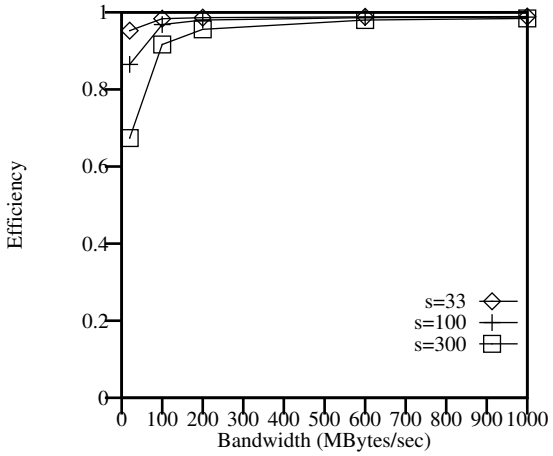

Figure 10: FFT

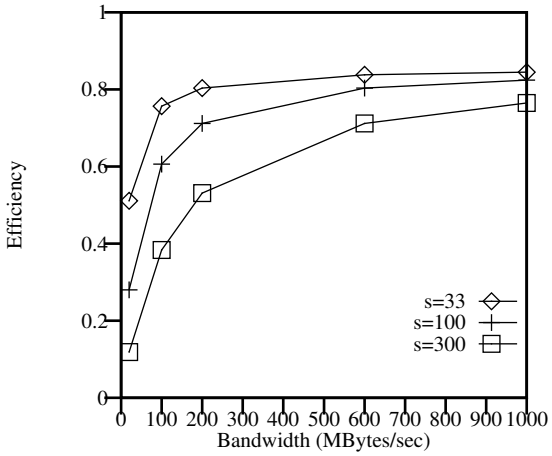

Figure 11: CG

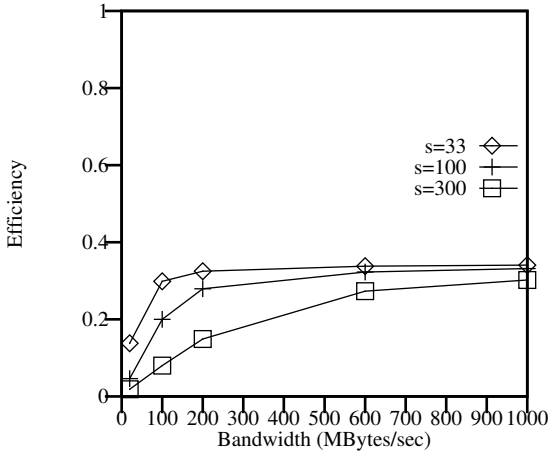

Figure 12: CHOLESKY

Impact of Problem Size

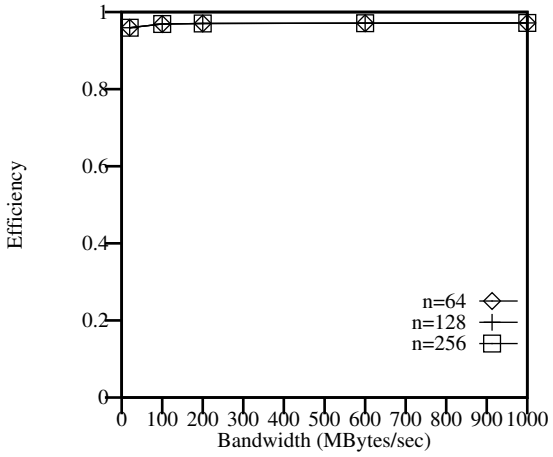

Figure 13: EP

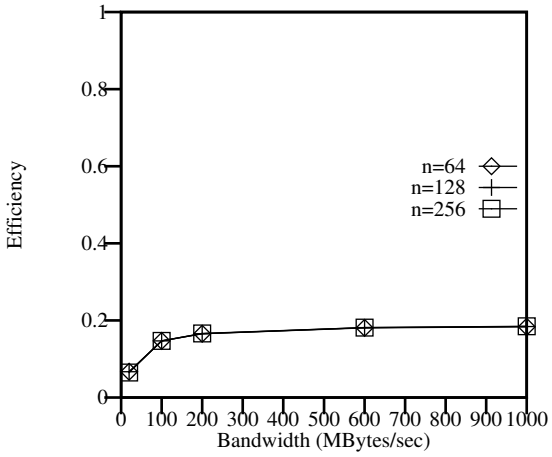

Figure 14: IS

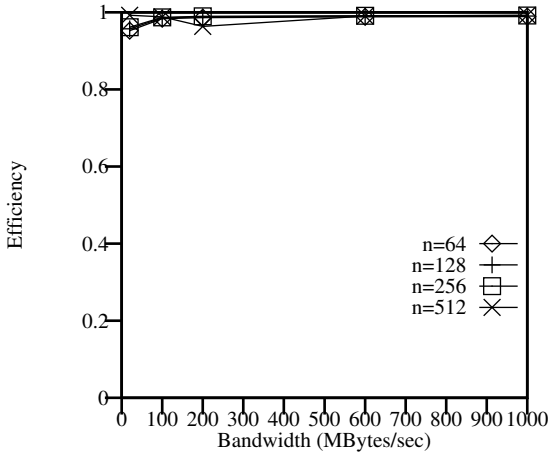

Figure 15: FFT

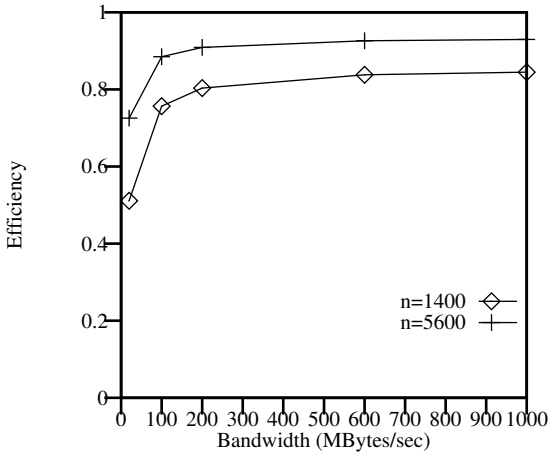

Figure 16: CG

Bandwidth Requirements of EP

Overheads	p=4	p=8	p=16	p=32	p=64	Bandwidth Functions	p=1024
50%	< 1.0	< 1.0	< 1.0	< 1.0	< 1.0	-	-
30%	< 1.0	< 1.0	< 1.0	< 1.0	< 1.0	-	-
10%	< 1.0	< 1.0	< 1.0	< 1.0	1.17	-	-

n=128K, s=33 MHz

Table 1: EP: Impact of Processors on Link Bandwidth (in MBytes/sec)

Overheads	s=33MHz	s=100MHz	s=300MHz
50%	0.54	1.69	4.68
30%	0.90	2.81	7.80
10%	1.17	3.67	10.41

p=64, n=128K

Table 2: EP: Impact of CPU speed on Link Bandwidth (in MBytes/sec)

Bandwidth Requirements of IS

Overheads	p=4	p=8	p=16	p=32	p=64	Bandwidth Functions	p=1024
50%	7.75	13.38	22.00	38.65	47.03	$23.60p^{0.28} - 28.79$	143.25
30%	12.91	30.75	66.44	78.61	84.61	$74.41p^{0.22} - 91.21$	251.91
10%	68.69	92.87	168.71	211.45	293.44	$88.68p^{0.34} - 82.12$	907.80

n=64K, s=33 MHz

Table 3: IS: Impact of Processors on Link Bandwidth (in MBytes/sec)

Overheads	s=33MHz	s=100MHz	s=300MHz
50%	47.03	102.49	356.14
30%	84.61	224.69	649.95
10%	293.44	770.16	1344.72

p=64, n=64K

Table 4: IS: Impact of CPU speed on Link Bandwidth (in MBytes/sec)

Overheads	n=16K	n=32K	n=64K	n=128K	n=256K	Bandwidth Functions	n=8192K
50%	46.60	47.16	47.03	47.48	48.67	$0.007n^{1.00} + 46.61$	110.75
30%	83.80	84.52	84.61	85.09	85.53	$0.006n^{0.99} + 84.10$	133.19
10%	270.08	286.98	293.44	303.41	307.75	$19.57n^{0.26} + 230.19$	441.66

p=64, s=33 MHz

Table 5: IS: Impact of Problem Size on Link Bandwidth (in MBytes/sec)

Overheads	s=33MHz	s=100MHz	s=300MHz
50%	337.34	735.15	> 5000
30%	396.55	1053.08	> 5000
10%	1366.34	3586.08	> 5000

$p = 1024$, $n = 2^{23}$

Table 6: IS: Link Bandwidth Projections (in MBytes/sec)

Bandwidth Requirements of FFT

Overheads	p=4	p=8	p=16	p=32	p=64	Bandwidth Functions	p=1024
50%	< 1.0	< 1.0	< 1.0	< 1.0	< 1.0	-	-
30%	6.40	6.52	7.52	7.83	8.65	$0.75p^{0.36} + 5.11$	14.85
10%	16.35	16.40	16.75	16.87	17.19	$0.01p^{0.99} + 16.37$	29.93

n=64K, s=33 MHz

Table 7: FFT: Impact of Processors on Link Bandwidth (in MBytes/sec)

Overheads	s=33MHz	s=100MHz	s=300MHz
50%	< 1	8.65	17.19
30%	8.65	13.81	29.20
10%	17.19	29.86	88.81

p=64, n=64K

Table 8: FFT: Impact of CPU speed on Link Bandwidth (in MBytes/sec)

Overheads	n=16K	n=32K	n=64K	n=128K	n=256K	Bandwidth Functions	n=2^{30}
50%	< 1	< 1	< 1	< 1	< 1	-	-
30%	9.42	9.03	8.65	8.38	7.97	$11.02 - 0.4 \log n$	3.02
10%	17.63	17.45	17.19	17.03	16.84	$18.43 - 0.2 \log n$	14.43

p=64, s=33 MHz

Table 9: FFT: Impact of Problem Size on Link Bandwidth (in MBytes/sec)

Overheads	s=33MHz	s=100MHz	s=300MHz
50%	-	-	-
30%	5.15	8.22	17.38
10%	21.64	48.58	144.50

$p = 1024$, $n = 2^{30}$

Table 10: FFT: Link Bandwidth Projections (in MBytes/sec)

Bandwidth Requirements of CG

Overheads	p=4	p=8	p=16	p=32	p=64	Bandwidth Functions	p=1024
50%	1.74	3.25	5.81	9.73	15.63	$1.25p^{0.62} - 1.28$	94.79
30%	2.90	5.41	9.68	16.22	46.10	$0.04p^{1.63} + 3.61$	393.32
10%	8.71	16.23	52.03	82.39	124.19	$18.80p^{0.51} - 33.07$	618.28

n=1400*1400, s=33 MHz

Table 11: CG: Impact of Processors on Link Bandwidth (in MBytes/sec)

Overheads	s=33MHz	s=100MHz	s=300MHz
50%	15.63	43.50	120.89
30%	46.10	96.75	262.84
10%	124.19	386.12	1022.14

p=64, n=1400*1400

Table 12: CG: Impact of CPU speed on Link Bandwidth (in MBytes/sec)

Overheads	n=1400*1400	n=5600*5600
50%	15.63	9.48
30%	46.10	25.47
10%	124.19	78.55

p=64, s=33 MHz

Table 13: CG: Impact of Problem Size on Link Bandwidth (in MBytes/sec)

Overheads	s=33MHz	s=100MHz	s=300MHz
50%	34.87	97.05	269.7
30%	120.04	251.93	684.41
10%	247.33	1200.49	2035.64

$p = 1024$, $n = 14000 * 14000$

Table 14: CG: Link Bandwidth Projections (in MBytes/sec)

Bandwidth Requirements of CHOLESKY

Overheads	p=4	p=8	p=16	p=32	p=64	Bandwidth Functions	p=1024
50%	5.62	6.86	7.77	8.91	10.49	$1.47p^{0.37} + 3.43$	23.44
30%	11.98	13.11	14.48	16.02	17.48	$2.14p^{0.33} + 8.82$	31.26
10%	76.56	78.32	80.83	84.12	87.35	$6.77p^{0.27} + 66.60$	110.70

n=1806*1806, s=33 MHz

Table 15: CHOLESKY: Impact of Processors on Link Bandwidth (in MBytes/sec)

Overheads	s=33MHz	s=100MHz	s=300MHz
50%	10.49	16.56	29.51
30%	17.48	60.13	171.29
10%	87.35	278.92	712.60

p=64, n=1806*1806

Table 16: CHOLESKY: Impact of CPU speed on Link Bandwidth (in MBytes/sec)

Appendix

EP

Phase	Description	Comp. Gran.	Data Gran.	Synchronization
1	Local Float. Pt. Opns.	Large	N/A	N/A
2	Global Sum	Integer Add	Integer	Wait-Signal

Table 17: Characteristics of EP

EP is an "Embarrassingly Parallel" application that generates pairs of Gaussian random deviates and tabulates the number of pairs in successive square annuli. This problem is typical of many Monte-Carlo simulation applications. It is computation bound and has little communication between processors. A large number of floating point random numbers n is generated which are then subject to a series of operations. The computation granularity of this section of the code is considerably large and is linear in the number of random numbers (the problem size) calculated. The operation performed on a computed random number is completely independent of the other random numbers. The processor assigned to a random number can thus execute all the operations for that number without any external data. Hence the data granularity is meaningless for this phase of the program. Towards the end of this phase, a few global sums are calculated by using a logarithmic reduce operation. In step i of the reduction, a processor receives an integer from another which is a distance 2^i away and performs an addition of the received value with a local value. The data that it receives (data granularity) resides in a cache block in the other processor, along with the synchronization variable which indicates that the data is ready (synchronization is combined with data transfer to exploit spatial locality). Since only 1 processor writes into this variable, and the other spins on the value of the synchronization variable (Wait-Signal semantics), no locks are used. Every processor reads the global sum from the cache block of processor 0 when the last addition is complete. The computation granularity between these communication steps can lead to work imbalance since the number of participating processors halves after each step of the logarithmic reduction. However since the computation is a simple addition it does not cause any significant imbalance for this application. The amount of local computation in the initial computation phase overshadows the communication performed by a processor. Table 17 summarizes the characteristics of EP.

IS

IS is an "Integer Sort" application that uses bucket sort to rank a list of integers which is an important operation in "particle method" codes. An implementation of the algorithm is described in [20] and Table 18 summarizes its characteristics. The input list of size n is equally partitioned among the processors. Each processor maintains two sets of buckets. One set of buckets (of size $nbuckets$) is used to maintain the information for the portion of the list local to it. The other set (of size $chunk$ = $nbuckets/p$ where p is the number of processors) maintains the information for the entire list. A processor first updates the local buckets for the portion of the list allotted to it, which is an entirely local operation (phase 1). Each list element would require an update (integer addition) of its corresponding bucket. A barrier is used to ensure the completion of this phase. The implementation of the barrier is similar to the implementation of the logarithmic global sum

Phase	Description	Comp. Gran.	Data Gran.	Synchronization
1	Local bucket updates	Small	N/A	N/A
2	Barrier Sync.	N/A	N/A	Barrier
3	Global bucket merge	Small	$chunk * (p-1)$ integers	N/A
4	Global Sum	Integer Add	Integer	Wait-Signal
5	Global bucket updates	Small	N/A	N/A
6	Barrier Sync.	N/A	N/A	Barrier
7	Global bucket updates	Small	$nbuckets$ integers	Lock each bucket
8	Local List Ranking	Small	N/A	N/A

Table 18: Characteristics of IS

operation discussed in EP, except that no computation need be performed. A processor then uses the local buckets of every other processor to calculate the bucket values for the $chunk$ of the global buckets allotted to it (phase 3). The phase would thus require $chunk * (p-1)$ remote bucket values per processor. During this calculation, the processor also maintains the sum of all the global bucket values in its $chunk$. These sums are then involved in a logarithmic reduce operation (phase 4) to obtain the partial sum for each processor. Each processor uses this partial sum in calculating the partial sums for the $chunk$ of global buckets allotted to it (phase 5) which is again a local operation. At the completion of this phase, a processor sets a lock (test-test&set lock [4]) for each global bucket, subtracts the value found in the corresponding local bucket, updates the local bucket with this new value in the global bucket, and unlocks the bucket (phase 7). The memory allocation for the global buckets and its locks is done in such a way that a bucket and its corresponding lock fall in the same cache block and the rest of the cache block is unused. Synchronization is thus combined with data transfer and false sharing is avoided. The final list ranking phase (phase 8) is a completely local operation using the local buckets in each processor and is similar to phase 1 in its characteristics.

FFT

Phase	Description	Comp. Gran.	Data Gran.	Synchronization
1	Local radix-2 butterfly	$O(\frac{n}{p}\log\frac{n}{p})$	N/A	N/A
2	Barrier Sync.	N/A	N/A	Barrier
3	Data redistribution	N/A	$(p-1) * \frac{n}{p^2}$complex numbers	N/A
4	Barrier Sync.	N/A	N/A	Barrier
5	Local radix-2 butterfly	$O(\frac{n}{p}\log p)$	N/A	N/A

Table 19: Characteristics of FFT

FFT is a one dimensional complex Fast Fourier Transform of n points that plays an important role in Image and Signal processing. n is a power of 2 and greater than or equal to the square of the number of processors p. There are three important phases in the application. In the first and last phases, processors perform the radix-2 butterfly computation on n/p local points. The only communication is incurred in the middle phase in which the *cyclic* layout of data is changed to a *blocked* layout as described in [8]. It involves an all-to-all communication step where each processor distributes its local data equally among the p processors. The communication in this step is *staggered* with processor i starting with data ($\frac{n}{p^2}$

points) read from processor $i+1$ and ending with data read from processor $i-1$ in $p-1$ substeps. This communication schedule minimizes contention both in the network and at the processor ends. These three phases are separated by barriers.

CG

Phase	Description	Comp. Gran.	Data Gran.	Synchronization
1	Matrix-Vector Prod.	Medium	Random Float. Pt. Accesses	N/A
2	Vector-vector Prod.			
	a) Local dot product	Small	N/A	N/A
	b) Global Sum	Float. Pt. Add	Float. Pt.	WaitSignal
3	Local Float. Pt. Opns	Medium	N/A	N/A
4	<same as phase 2>			
5	Local Float. Pt. Opns	Medium	N/A	N/A
6	Barrier Sync.	N/A	N/A	Barrier

Table 20: Characteristics of CG

CG is a "Conjugate Gradient" application which uses the Conjugate Gradient method to estimate the smallest eigenvalue of a symmetric positive-definite sparse matrix with a random pattern of non-zeroes that is typical of unstructured grid computations. The sparse matrix of size $n * n$ and the vectors are partitioned by rows assigning an equal number of contiguous rows to each processor (static scheduling). We present the results for five iterations of the Conjugate Gradient Method in trying to approximate the solution of a system of linear equations. There is a barrier at the end of each iteration. Each iteration involves the calculation of a sparse matrix-vector product and two vector-vector dot products. These are the only operations that involve communication. The computation granularity between these operations is linear in the number of rows (the problem size) and involves a floating point addition and multiplication for each row. The vector-vector dot product is calculated by first obtaining the intermediate dot products for the elements in the vectors local to a processor. This is again a local operation with a computation granularity linear in the number of rows assigned to a processor with a floating point multiplication and addition performed for each row. A global sum of the intermediate dot products is calculated by a logarithmic reduce operation (as in EP) yielding the final dot product. For the computation of the matrix-vector product, each processor performs the necessary calculations for the rows assigned to it in the resulting matrix (which are also the same rows in the sparse matrix that are local to the processor). But the calculation may need elements of the vector that are not local to a processor. Since the elements of the vector that are needed for the computation are dependent on the randomly generated sparse matrix, the communication pattern for this phase is random. Table 20 summarizes the characteristics for each iteration of CG.

CHOLESKY

This application performs a Cholesky factorization of a sparse positive definite matrix of size $n * n$. The sparse nature of the input matrix results in an algorithm with a data dependent dynamic access pattern. The algorithm requires an initial symbolic factorization of the input matrix which is done sequentially because it requires only a small fraction of the total compute time. Only numerical factorization [25] is parallelized and analyzed. Sets of columns having similar non-zero

Phase	Description	Comp. Gran.	Data Gran.	Synchronization
1	Get task	integer addition	few integers	mutex lock
2	Modify supernode	supernode size float. pt. ops.	supernode	N/A
3	Modify s supernodes (s is data dependent)	$s*$ supernode size float. pt. ops	s supernodes	locks for each column
4	Add task (if needed)	integer addition	few integers	lock

Table 21: Characteristics of CHOLESKY

structure are combined into supernodes at the end of symbolic factorization. Processors get tasks from a central task queue. Each supernode is a potential task which is used to modify subsequent supernodes. A *modifications_due* counter is maintained with each supernode. Thus each task involves fetching the associated supernode, modifying it and using it to modify other supernodes, thereby decreasing the *modifications_due* counters of supernodes. Communication is involved in fetching all the required columns to the processor working on a given task. When the counter for a supernode reaches 0, it is added to the task queue. Synchronization occurs in locking the task queue when fetching or adding tasks, and locking columns when they are being modified.